Fuzzing for fun and for $$$
« hola mujer, what’s that fuzzy noise over the wire? »

• Lecturers:
  • S. Bekrar\textsuperscript{1}
  • Fabien Duchene\textsuperscript{1}

\textsuperscript{1} LIG Lab, VASCO team
Outline

0. Software security: Why?

1. How to check the security of a system?

2. Testing

3. Vulnerability discovery

4. Some vulnerabilities

5. Fuzzing

6. Hands on lab!
   - File parser fuzzing
   - Protocol fuzzing
   - Evolutionary fuzzing
   - WhiteBox Fuzzing

7. New trends in fuzzing
   -- symbolic execution
   -- evolutionary fuzzing
“Anyone can find a new vulnerability in almost any piece of commercial software and write an exploit for it. Anyone can even write worms or viruses with some focus on the topic.”

=> Systems and their users are exposed to attackers

If an attacker is allowed to execute malicious code on the system then system confidentiality, integrity, and availability are compromised
1. How to check the security of a system?

- **Formal Proof, Verification**
  - Type-checking
  - Model-checking
  => Systematic, exhaustive
  But: False positives, time-consuming, not scalable

- **Code analysis methods**
  - Source code auditing
  - Reverse code engineering
  => Systematic, exhaustive, scalable
  But: False positives, time-consuming

- **Interface testing methods** *(Test)*
  - Penetration testing
  - Fuzzing
  => Real errors, fast
  But: Poor coverage, more errors missed
2. Testing

2.1. Why?

2.2. Vocabulary:
   - SUT
   - Interfaces
   - Hypotheses
   - Fault
   - Coverage

2.3. Testing taxonomy
   - Passive, active
   - Knowledge, access
   - Development stage
   - Test set creation method

2.4. Some testing methodologies
   - Test Driven Development
   - Mutation based testing
2.1. Testing: WHY?

**Software Development LifeCycle (SDLC):**

- Testing budget accounts from 50% of total till 80% in critical sectors (transportation)

- The later a bug is discovered, the more costly for the software vendor
  - 6,000$ - 9,000$ cost when early steps

---

2.1. Testing: WHY? (2)

**FUZZING for Software Security Testing and Quality Assurance**, A. Takanen, Jared D. Demott, Charlie Miller
• **Testing != Debugging**
  • At first, not interested in finding the cause
  • Nor correcting faults

• System Under Test (SUT)
Some testing challenges

- **Repeatability:**
- **Verdict:**
  - Oracle, how to detect the fault? (=> monitoring)
- **Coverage.** Some examples:
  - how much was this **feature** stressed?
  - How many **percent of the SUT** did we test:
    - Path
    - Nodes
  - How many faults
- **Duration**
Fault is the r00t my lord!

• **Fault, Bug, Defect**
  - **Fault**: portion of a SUT not correctly written
    - = vulnerability in security context
  - **Error**: difference between computed value and the correct one
  - **Failure (défaillance)**:
    - the error is repercuted in the outputs
  - **Breakdown (panne)**:
    - Security property broken

Ensimag-5MMTLSFT-Software Testing, 2011, R. Groz
Test Oracle

- Provides **verdict:**
  - **Pass** ✓
    - No bug detected
  - **Fail** ✗
    - Bug detected
  - **Inconclusive** ?
    - Cannot tell
**Coverage**

Measure/metric “how MUCH of …. has been tested”

Some coverage metrics examples:

<table>
<thead>
<tr>
<th>Coverage metric</th>
<th>White-Box</th>
<th>Grey-Box</th>
<th>Black-Box</th>
<th>Constraints</th>
<th>Tool ex.</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input mutation</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lines of code</td>
<td>X</td>
<td>X</td>
<td></td>
<td>recompilation</td>
<td>gcov</td>
<td></td>
</tr>
<tr>
<td>Function Entry/Exit</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>Debuggers: IDA, Immunity...</td>
<td></td>
</tr>
<tr>
<td>Loop</td>
<td>X</td>
<td>X</td>
<td></td>
<td>Binary instrumentation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic Block</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Path</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>n decisions, =&gt; n conditional jumps =&gt; (2^n) paths</td>
<td></td>
</tr>
</tbody>
</table>
Some Testing Hypotheses

- Any system can be modeled as an IOLTS
- Uniform partitionig:
  
  \[ E = A \oplus B \oplus C \]

  \[ \forall a_i \in A, \ [\text{success}(P(a_i)) \iff \forall x_i \in A \ \text{success}(P(x_i))] \]
Testing taxonomy

• Tests can be categorized according different viewpoints:
  • Properties to be validated
  • Development stage
  • SUT knowledge + access
  • Test set creation method
  • Tested Object
Testing taxonomy – properties to be validated

<table>
<thead>
<tr>
<th>Positive / functional</th>
<th>Negative / non functional</th>
</tr>
</thead>
<tbody>
<tr>
<td>If we send this and that inputs, the SUT should output $O_1, O_2$</td>
<td>No matter ANY inputs we sent to the SUT, that property is NEVER violated.</td>
</tr>
<tr>
<td><em>Eg:</em> IF I enter my contact details and the number and types of items I want on Pedro-Pizza, THEN the checkout page is presented.</td>
<td><em>Eg:</em></td>
</tr>
<tr>
<td><strong>Fuzzing</strong></td>
<td><strong>Bad behavior in pgm</strong> (e.g., assertions violated)</td>
</tr>
<tr>
<td><strong>Robustness</strong></td>
<td><strong>Attacker can benefit of weakness in behavior</strong> (assertions generally OK)</td>
</tr>
<tr>
<td>Performance: - ability to serve XX users</td>
<td>Performance: - response time ALWAYS below XXX seconds</td>
</tr>
</tbody>
</table>
Testing taxonomy - Passive vs Active

- **Passive:**
  - detect invariant properties violation on traces

- **Active:**
  - Actively submit (new) inputs to the SUT
Testing taxonomy – by knowledge / access

- SUT related **knowledge / access / visibility:**
  - **BlackBox:**
    - Only **interface knowledge** assumed
  - **GreyBox:**
    - Partial knowledge assumed
    - Eg: access to assembly code
  - **WhiteBox:**
    - Complete Knowledge about the **SUT internals** assumed
    - Eg: access to source code
## Testing taxonomy – by SUT Interfaces

<table>
<thead>
<tr>
<th>SUT</th>
<th>Executable examples</th>
<th>Some INPUT testing interfaces</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol server</td>
<td>proFTPD</td>
<td>Sockets, logs</td>
<td>Socket, filesystem</td>
</tr>
<tr>
<td>File parser</td>
<td>cat</td>
<td>Files, keystrokes</td>
<td>stdout</td>
</tr>
<tr>
<td>Drawing tool</td>
<td>Paint.Net</td>
<td>Mouse clicks + moves, keystrokes + file reads n writes</td>
<td>Drawable window</td>
</tr>
</tbody>
</table>
Testing taxonomy – by development stage

• **Early development stages**: Unit test
  • Adequacy criterion

• **Before release**: Alpha, Beta, Gamma tests
  • Performed inside Microsoft Corp. before releasing to the customers
  • Scenarios

• **Integration**:  
  • several components tested together  
    o Bottom-up  
    o Top-down
Testing taxonomy – by test set creation method

How is the test set / test campaign created?

- Unit testing
- Model Based Testing
- Scenario
• A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.
3. Vulnerability discovery: WHY?

• Although the introduction of security techniques into software development life-cycle, we still discover many vulnerabilities
• CVE …
4. Some Low Level Vulnerabilities

Memory Corruption
- Stack-Based Exploitation for
  - Buffer Over-Flow (TP2)
  - Format String
- Heap overflow
- Use-After-Free
- Double Free

Command Injection
- Cross Site Scripting (XSS) (TP1)
- SQL Injection
- Shell Command Injection
- PHP Code injection
5. Fuzzing

<table>
<thead>
<tr>
<th>5.1. What?</th>
<th>5.6. Who should fuzz?</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2. Why?</td>
<td>5.7. Who fizzes?</td>
</tr>
<tr>
<td>5.3. Fuzzing LifeCycle</td>
<td>5.8. Some fuzzers</td>
</tr>
<tr>
<td>5.4. How?</td>
<td>5.9. Some references</td>
</tr>
<tr>
<td>5.5. Fuzzing classification</td>
<td></td>
</tr>
</tbody>
</table>
5. Fuzzing

- test generation technique for discovering vulnerabilities

- The key idea: feeding malformed inputs to an application
  - Malformed?
    - RFC?
    - Software implementation?

- Goal: discovering IMPLEMENTATION faults ..
  - Not logical protocol flaws

- A common fault model:
  - SUT crash or hang
    - Intuition: it might be a vulnerability and might be exploitable
  - Clearly not the only fault we can think of, but simple
5.1 Fuzzing: WHAT?

• Once upon a time…
  • Barton Miller 1988, storm, noises over line wire:
    o Difficulty to type commands
    o Crash daily used executables
    o University student project: random unstructured data “fuzz”
5.2. Fuzzing: WHY?

- Low cost
- Fast
- Find real errors (less false negatives)
- Even errors with **high impacts**:
  - From Denial Of Service (DOS) till Remote Code Execution + Elevation of Privilege
5.3. Fuzzing life-cycle

- Identifying interfaces
- Generating fuzzed data
- Feeding fuzzed data to the target
- Monitoring and identifying faults
- Exploitability evaluation
- Reporting (or not)
5.4. HOW to fuzz?

- E.g., Mutation Based Fuzzing:
  
  Charlie Miller:
  - Extract seed files from the web
  - Mutate them
  - Eventually Crossover
  - Submit
HOW to fuzz?

- The equivalence partitioning hypothesis is often associated with boundary value testing. Eg:
  - **Integer overflow**: testing for several n, $2^n$
5.5. Fuzzing classification: how to fuzz?

- **Random**
  - Generating tests randomly

- **Mutation-based**
  - Applying mutations on existing data

- **Generation based**
  - Make of an input Grammar

- **Model-based**
  - Modelling the target protocol/file format
5.6. Some methods - Random/blind/dumb fuzzing

- cat /dev/random > [sut]
- **Dumb**: does not respect specifications (might even not be aware)
- ... might be interesting for a company depending on the complexity the software they test (cost, time, results..)

State 1

- Deeply embedded bugs will very probably not be found (conditions/guards on transitions)

+ fast

- easy

S2

S2.1

S2.2

S3
5.6. Some fuzzing methods - Mutation Based Fuzzing

Valid input sequence (eg: wireshark capture, valid PNG file)

- `<INPUT ... alt="hola" >`
- `hola`: fuggable=True

Fuzzed test case

- `<INPUT ... alt="whtf::ASLKKLSA %d00-1.zkshajaksd?’../..” />

At least one mutation operator applied to at least one "token"

The applied mutation operator does not seem efficient. IMHO: better target 1 vulnerability at a time.

Eg:
- BOF: **9000x"d”**
- Int. UnderFlow: -1
- XSS: ” `onmouseover=”alert(‘cuicui’);”>`
5.6. Some methods - Generation Based Fuzzing

- For at least one input parameter, a grammar is written

```
HTML_XSS_FIELD ::= HTML_TEXT_SIMPLE HTML_TAG_QUOTE
HTML_TAG_SPACE HTML_TAG_EVENT HTML_TAG_EQUAL
HTML_TAG_QUOTE JS_PAYLOAD

HTML_TAG_QUOTE ::= ’ | ”
HTML_TAG_SPACE ::= \n | \t | \r |
HTML_TAG_EQUAL ::= =
HTML_TAG_EVENT ::= onabort | … | onclick | … | onwaiting
```

- Part of a test case is a WORD produced by that grammar

```
kalimu " \n onclick = " alert(1)
```
5.6. Some methods - Model Based Fuzz testing

- **Requirement:** **SUT model** (might be abstract)
  - Manually written, or **inferred** (K. Li, R. Groz, K. Hossen, C. Oriat)

```
S0  i_1^a  S1
    /     |
  i_1^b  q = ...<a name="kalimu">Hola!</a>... q_1
  |     / ~~~~~
  |    q_2
S2  i_1^b  S3
```

- **Some mutation operations:**
  - Changing an input parameter value (e.g. $i_1^3$)
  - Sending a fuzzed input parameter value for another transition (e.g. for $i_1^2$ we send $\text{mutation\_op}(i_1^4)$)
### Some fuzzer categories comparison

<table>
<thead>
<tr>
<th></th>
<th>Random fuzzing</th>
<th>Mutation-based fuzzing</th>
<th>Model-based fuzzing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pros</strong></td>
<td>• Easy to implement</td>
<td>• Relatively inexpensive to implement</td>
<td>• Finding different error types</td>
</tr>
<tr>
<td></td>
<td>• Quick</td>
<td>• Theoretically effective</td>
<td>• Potentially efficient (if the target is well modeled)</td>
</tr>
<tr>
<td></td>
<td>• Low cost</td>
<td>• Reusable across different software</td>
<td></td>
</tr>
<tr>
<td><strong>Cons</strong></td>
<td>• Only the target surfaces attacked</td>
<td>• Efficiency depends on the number of valid inputs available</td>
<td>• Time consuming to set up</td>
</tr>
<tr>
<td></td>
<td>• Poor coverage</td>
<td></td>
<td>• Knowledge of the format/protocol is required</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Reusable only with same format</td>
</tr>
</tbody>
</table>
5.6. Fuzzer classification: where to fuzz?

- **Local fuzzers**
  - Command line fuzzers
  - Environment variable fuzzers
  - File format fuzzers

- **Remote fuzzers**
  - Network protocol fuzzers
  - Web application fuzzers
  - Web browser fuzzers

- **In-memory fuzzers**
  - Simple protocol
  - Complex protocol
5.7. WHO should fuzz?

- Developers
- QA analysts
- Security researchers

- Design
- Implementation
- Quality assurance
- Production
5.8. WHO fuzzes?

- **Microsoft** - Patrice Godefroid, Adam Kiezun, and Michael Y. Levin
  - Grammar-based Whitebox Fuzzing, 2007

- **Google** - Tavis Ormandy
  - Flayer: Exposing Application Internals, 2007

- **Charlie Miller**:
  - Fuzzing the Phone in your Phone, 2009

- **Orange R&D** - Laurent Butti
  - Wifi advanced fuzzing, 2007
  - several CVE in 802.11 drivers
### Historical fuzzers

<table>
<thead>
<tr>
<th>Fuzzer name</th>
<th>year</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet Stack Integrity Checker (ISIC)</td>
<td>1999</td>
<td>IP, TCP, UDP, ICMP, .. Protocols implementations</td>
</tr>
<tr>
<td>SPIKE (Protos)</td>
<td>2002</td>
<td></td>
</tr>
</tbody>
</table>
## Some fuzzers

<table>
<thead>
<tr>
<th>Name</th>
<th>OS</th>
<th>Type of target</th>
<th>Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEACH</td>
<td>LINUX, WINDOWS</td>
<td>File Format, Network protocol</td>
<td>Meta-modelling</td>
</tr>
<tr>
<td>FUZZGRIND</td>
<td>LINUX</td>
<td>File Format</td>
<td>Symbolic Execution</td>
</tr>
<tr>
<td>SULLEY</td>
<td>WINDOWS, LINUX</td>
<td>Network protocol</td>
<td>Learning Approach</td>
</tr>
<tr>
<td>AUTODAFE</td>
<td>LINUX</td>
<td>File format, Network protocol</td>
<td>Fuzzing by weighting attacks with markers</td>
</tr>
</tbody>
</table>
### Some fuzzers

<table>
<thead>
<tr>
<th>Name</th>
<th>OS</th>
<th>Type of target</th>
<th>Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>CATCHCONV</td>
<td>LINUX</td>
<td>File Format</td>
<td>Meta-modelling</td>
</tr>
<tr>
<td>BUNNY</td>
<td>LINUX</td>
<td>File Format</td>
<td>Symbolic Execution</td>
</tr>
<tr>
<td>CUTE, DART, EXE</td>
<td>LINUX</td>
<td>Network protocol File Format</td>
<td>Learning Approach</td>
</tr>
<tr>
<td>SAGE</td>
<td>WINDOWS</td>
<td>File format</td>
<td>Fuzzing by weighting attacks with markers</td>
</tr>
</tbody>
</table>
Fuzzing – some references

• FUZZING for Software Security Testing and Quality Assurance
  • Ari Takanen
    o Codenomicon
    o @aritakanen
  • Jared D. Demott
    o VDA Labs
    o @jareddemott
  • Charlie Miller
    o Accuvant Labs
    o @0xcharlie
Fuzzing – some references (2)

- FUZZING: Brute Force Vulnerability Discovery
  - Michael Sutton
    - Zscaler Threat Labs
    - @michaelawsutton
  - Adam Greene
  - Pedram Amini
    - DVLabs
    - @pedramamini
7. Hands on lab agent!

7.1. File Parser Fuzzing

7.2. BlackBox Model-Based Protocol Fuzzing

7.3. GreyBox Evolutionary Fuzzing
File Fuzzing: FileFuzz

FileFuzz: a graphical Windows-based fuzzer
Targets: file parsers
Approach: mutation

Three distinct phases:
• Creation of fuzzed files
• Execution of fuzzed files
• Monitoring the application (debugging functionalities)
FileFuzz: fuzzed data creation

- Read from a valid file
- Repeat:
  - Mutate specific segments according to provided directives:
    - Breadth:
      - All bytes: covering all bytes within a file
      - Range: changing byte values within a range to a predefined value until covering the entire range
    - Depth
    - Match
  - Save the generated file
FileFuzz: fuzzed data execution

- Determine how the application is launched from a command prompt
- Determine flags that should be passed to the application
  - Windows explorer
  - Tools…Folder Options…File Types
- Repeat:
  - Run the target with a new generated file
  - Monitor for exceptions
  - Kill the process if necessary
FileFuzz: exception detection

- Integrating a debugger into a fuzzer is important:
  - The fuzzer is launching and killing the target with each file
  - Manually attaching a debugger to the target = constantly killing the debugger
- crash.exe is a stand-alone debugger integrated into FileFuzz
- It automatically attaches to the target at each execution
- It monitors for exceptions
- It captures (if an exception is identified):
  - Memory location
  - Registry values
  - Exception type

```
[*] "crash.exe" "C:\Program Files\WordPerfect Office 12\Programs\UA120.exe" 2000 /qt c:\fuzz\ast\8.ast
[*] Access Violation
[*] Exception caught at 00403f06 mov eax,[eax+edi*4]
[*] EAX:0014b1b8 EBX:00000005 ECX:00435c00 EDX:0012fbac
[*] ESI:00435c00 EDI:cccccccc ESP:0012fab8 EBP:0012fae8
```
File Parser Fuzzing: FileFuzz
Model Based Testing

- “Automatable derivation of test cases from models and their execution”
An example of SUT model: FTP server
Sulley components architecture
BlackBox model based protocol fuzzing

• **Examples of coverage metrics:**
  - Number of reached states
  - Number of input variables tested for each transition

• **Oracle:**
  - XSS reflection on the very same transition = type 1 XSS
Fuzzing: some limitations

- Traditional dump BlackBox fuzzing
  - “Focuses more on the data than on state transitions”
- As always in testing: Non exhaustive
- A crash != a critical vulnerability
- Analyzing a crash: performed manually
- Complex vulnerabilities (based on a combination of vulnerabilities) are not detected
- Mutation based fuzzers can generate an infinite number of test cases. When to stop?
- Figuring out which test case caused the fault
- Finding the vulnerability from a given a crash
- Identifying the changes necessary to improve a fuzzer after fuzzing.
7. Some current research directions:

- **Evolutionary Fuzzing**
  - Jared D. Demott
  - F. Duchene, S. Rawat, J.L. Richier, R. Groz

- **Symbolic Execution**
  - Cristian Cadar

- **Model Based Fuzzing**
  - SPaCioS FP7 and ITEA2 Diamonds projects
Vulnerability discovery: Conclusions

- Security measures introduced in the development life-cycle are not enough:
  - No method can guarantee that there are no flaws left in the software
- Multiple techniques are better than one
- “The More Fuzzers The Better” C. Miller